UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

Aplicación móvil para la administración de recursos de transporte sanitario

Autor/es

PABLO RUBIO AMONDARAIN

Director/es

BEATRIZ PÉREZ VALLE

Facultad

Facultad de Ciencia y Tecnología

Titulación

Grado en Ingeniería Informática

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2018-19

publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

UNIVERSIDAD DE LA RIOJA

Facultad de Ciencia y Tecnología

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Aplicación móvil para la administración de recursos de transporte sanitario

Realizado por:

Pablo Rubio Amondarain

Tutelado por:

Beatriz Pérez Valle

Logroño, 24 de julio, 2019

Resumen

A lo largo de este trabajo de fin de grado veremos cómo se ha llevado a cabo el proyecto propuesto por Javier Virto, responsable de la empresa *GFI* en Logroño. El objetivo de este proyecto es desarrollar una aplicación *Android* que ayude a los tripulantes de una ambulancia a administrar los traslados de pacientes que deben realizar.

Este documento explica el proceso llevado a cabo para la realización del proyecto. Para ello, veremos cada una de las fases realizadas para su desarrollo, en qué consisten y cómo se han abordado.

Abstract

Through this dissertation I am going to present, how I have developed a project proposed by Javier Virto, the director of *GFI* Logroño. This project is about an application for an *Android* device, that helps the crew of an ambulance to manage the transfers of patients they must do.

This document explains the process I have followed to develop this project. Therefore, I will describe the phases I had completed to develop this project and how I have addressed them.

Índice general

Introducción

En la era digital, el mundo quiere llevar toda la información y funcionalidad posible a cualquier lugar, y si es en el bolsillo, mejor. Esto hace que el móvil juegue un papel fundamental en nuestro día a día, no solo por su cometido inicial, la comunicación, sino porque se ha convertido en un pequeño ordenador que llevamos con nosotros. Esto provoca que haya una gran demanda de desarrollo para aplicaciones móviles, de hecho, es preferible desarrollar muchos de los programas o aplicaciones para estos, ya que, en este momento, lideran sectores hasta ahora considerados para otros productos.

Hace unos meses realicé prácticas en GFI, una empresa multinacional de informática. La sede que se encuentra en Logroño trabaja sobre todo en el ambiente sanitario, ya sea público o privado. Mi responsable, Javier Virto, me propuso desarrollar el proyecto que expondré a continuación.

Una de las peticiones sin cubrir en los hospitales de La Rioja es una aplicación móvil orientada a dispositivos *Android* para ser utilizada por los tripulantes de ambulancias ordinarias, esto es, las que no son de emergencia, con el fin de poder administrar los traslados que estas realizan. Entre otras cosas, se pide que desde esta aplicación se pueda disponer de la información de los *traslados* que se deben realizar. En este proyecto entenderemos por traslados un conjunto de peticiones que comparten hora de recogida y dirección (si van dirección al hospital o dirección al domicilio). Las *peticiones*, como su nombre indica, son los encargos que realizan los médicos para traer al hospital a un paciente o llevarlo de vuelta a su hogar para que, de esta manera, la ambulancia pueda llevar (o traer de vuelta a casa) a varias personas por traslado, realizando una ruta concreta.

La aplicación móvil también debe poder identificar a los pacientes con tan solo tomar una foto de su *Tarjeta Sanitaria* (TS). Por último, la aplicación debe enviar al hospital regularmente (aproximadamente cada minuto) la localización de la ambulancia, con el fin de que desde este se pueda estimar cuándo estará de regreso.

La empresa me propuso que desarrollase esta aplicación: la parte del servidor y la del cliente *Android*. El servidor mencionado tiene que proporcionar los servicios mínimos para que la aplicación *Android* cumpla con su cometido, entre ellos, permitirle conectarse a una base de datos (BD) y ofrecer los datos necesarios a la aplicación móvil. En la BD mencionada, se debe almacenar información acerca de pacientes, traslados, ambulancias, peticiones y usuarios.

Capítulo 1

Planificación

En este capítulo veremos la planificación que se ha seguido a lo largo de todo el proyecto. Para ello, identificaremos las distintas fases del proyecto, en qué orden se desarrollarán y cuánto tiempo estimamos que necesitaremos para realizar cada una.

1.1 Metodología que se ha utilizado

En esta sección, estudiaremos las distintas metodologías que se analizaron y el motivo por el que me decanté por una de ellas. En particular, consideré estas dos opciones desde el principio.

- La primera que consideré fue utilizar la metodología en *cascada*. Por la duración del Trabajo Fin de Grado (TFG) (300 horas), parecía la más adecuada. Esta se basa principalmente en recorrer las fases de planificación, análisis, diseño y desarrollo una única vez. Esta metodología requiere que, durante el análisis y diseño, se tenga que trabajar considerando todo el proyecto.

- La segunda opción es alguna metodología cíclica, como por ejemplo *Sprint*. La razón por la que consideré utilizar una de estas metodologías, fue porque en este proyecto se podían diferenciar dos fases: el servidor y el cliente *Android*. En el primer *Sprint* analizaría, diseñaría y desarrollaría el *software* que está en la parte servidor, y en el segundo *Sprint*, la parte *Android*.

Tras estudiar ambas alternativas, decidí seleccionar la primera: la metodología en *cascada*. Aunque se podían diferenciar fácilmente 2 *sprints*, el desarrollo de la parte servidor y la parte *Android*, la gran relación que hay entre ellas hizo que me decantase por analizar, diseñar y desarrollar ambas fases a la vez.

1.2 Identificación y descripción de las tareas

En esta sección analizaremos las distintas fases por las que pasé mientras desarrollaba el proyecto y las iré describiendo. Las fases mencionadas son:

- **Planificación:** A lo largo de esta tarea se identifican las fases y subfases a seguir para la realización del proyecto. A posteriori, se estima el tiempo que costará realizar cada una de estas fases.
- **Análisis:** En esta sección se analiza el problema, esto es, se identifica qué es exactamente lo que se tiene que desarrollar, qué condiciones mínimas debe de cumplir la aplicación y cómo las va a cumplir. Entre otras cosas, qué tecnologías y recursos materiales se deben utilizar.
- **Diseño:** En este apartado se diseñan las entidades de *dominio* que necesite el servidor, los métodos que deban permitir acceder a los servicios del servidor, etc. También veremos el prototipo de interfaz de la aplicación móvil y el *dominio* y *persistencia* que necesite esta. Finalmente se diseñan los test que se realizarán para verificar que el producto es correcto.
- **Desarrollo:** Se desarrolla lo diseñado en el anterior apartado. Una vez desarrollado se ejecutan las pruebas planificadas, si todo es correcto, se despliega el producto.
- **Aprendizaje y estudio del entorno:** La mayoría de tecnologías que se van a utilizar en este proyecto son nuevas o tienen alguna diferencia considerable con lo visto a lo largo de la carrera, como por ejemplo, el uso del Reconocimiento Óptico de Caracteres (OCR) o el uso de *Android Studio* para desarrollar una aplicación *Android*, claro está, si no se ha visto previamente la asignatura de *Informática Móvil*, como es el caso.
- **Seguimiento y Control:** Durante el proyecto, en ciertas fechas marcadas, se irá observando en qué situación se encuentran las tareas: si el tiempo que había estimado es similar al real, si las fechas de fin de secciones se van cumpliendo, etc.
- **Redacción del documento:** Esta sección no requerirá mucho tiempo, ya que la mayor parte de esta se documenta en otros apartados. Por ejemplo, en diseño, todos los documentos relacionados con esta parte estarán escritos utilizando las semanas estimadas para este apartado.

1.3 Estimación de tiempo

Cuando comencé este proyecto, me encontraba trabajando en GFI, finalizando los 48 créditos que me quedaban del Grado en Ingeniería Informática y realizando el Máster de Profesorado en la especialidad de Matemáticas en la Universidad de la Rioja. Esto hizo que la planificación fuese inestable, ya que disponía de escasas horas a la semana para dedicárselas al TFG y cualquier imprevisto o entregable de cualquier asignatura provocaría su paralización. Además, entre los meses de marzo y abril debía realizar las prácticas del máster durante todas las tardes en un instituto, lo cual limitó bastante mi dedicación a la realización del TFG.

Para la planificación de las tareas descritas en el apartado anterior, en un primer momento desarrollé una pequeña tabla de preanálisis referente al tiempo que estimaba tardar para cada fase. Para esta estimación de tiempo, tuve en cuenta que también estuve realizando el máster de profesorado, cuyo Trabajo Fin de Máster (TFM) tenía que entregar el 25 de junio. Por lo cual, la planificación tenía como fecha límite la convocatoria de julio.

Tarea	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**	**S10**
Planificación	■	■	■							
Análisis			■	■	■					
Diseño					■	■	■	■	■	■
Desarrollo										
Aprendizaje y estudio			■	■	■	■	■	■	■	■

Tabla 1.1: Diagrama de Gantt semanas 1-10 de la primera planificación

.

Tarea	**S11**	**S12**	**S13**	**S14**	**S15**	**S16**	**S17**	**S18**	**S19**	**S20**
Planificación										
Análisis										
Diseño	■	■	■							
Desarrollo			■	■	■	■				
Aprendizaje y estudio	■	■	■	■						

Tabla 1.2: Diagrama de Gantt semanas 11-20 de la primera planificación

.

En esta primera versión (Tablas 1.1 y 1.2), podemos observar que dos de los apartados identificados no se encuentran (Seguimiento y control, y Redacción del documento). Esto se debe a que en la primera planificación no los tomé en consideración.

En la semana 3 (4-11 de marzo), al empezar las prácticas del máster de profesorado, me vi sin tiempo para realizar el TFG. Por consecuencia, durante las semanas 3, 4, 5, 6 y 7 no pude avanzar el TFG hasta que decidí rescindir el contrato con GFI y dejar de trabajar. Por ello, retomé el proyecto nuevamente en la semana 8, a mediados de abril, rehaciendo la planificación.

Javier, por parte de la empresa, no tuvo ningún problema con que continuase con el proyecto que había sugerido él mismo, por lo cual, no tuve que cambiar ni de proyecto ni de tema.

Para la segunda planificación (Tablas 1.3 y 1.4), hay que tener en cuenta que la productividad en el mes de julio fue mucho mayor, una vez entregado el TFM y disponer de todo el tiempo para realizar el TFG. Por último, la mitad de la semana

9 y toda la semana 10 fueron Semana Santa y, aunque no tuve que realizar prácticas por la tarde, la productividad no aumentó ya que me dediqué a hacer la memoria de las prácticas del máster.

Tarea	**S1**	**S2**	**S3**	**S4**	**S5**	**S6**	**S7**	**S8**	**S9**	**S10**
Planificación	■	■						■		
Análisis		■							■	■
Diseño										
Desarrollo										
Aprendizaje y estudio										
Seguimiento y control										
Redacción del documento										

Tabla 1.3: Diagrama de Gantt semanas 1-10 de la segunda planificación

.

Tarea	**S11**	**S12**	**S13**	**S14**	**S15**	**S16**	**S17**	**S18**	**S19**	**S20**
Planificación										
Análisis										
Diseño	■	■	■	■	■					
Desarrollo					■	■	■	■	■	■
Aprendizaje y estudio	■	■	■	■	■					
Seguimiento y control		■			■					■
Redacción del documento								■	■	■

Tabla 1.4: Diagrama de Gantt semanas 11-20 de la segunda planificación

.

Las nuevas tablas de planificación presentadas están modificadas en la semana 8 del proyecto. En estas tablas, la semana 1 (S1) representa la semana del 18 al 25 de febrero, al igual que en la tabla de la primera planificación. Las semanas 3, 4, 5, 6 y 7 están vacías, tal y como indico en la introducción de este apartado.

Las semanas 11, 12, 13, 14 y 15 representan las semanas de mayo. Para cuando estas finalizasen, el diseño debía estar totalmente finalizado y empezado el desarrollo. Las semanas 16, 17, 18, 19 y 20 representan junio y el comienzo de julio. Para principios de julio, el proyecto debería estar finalizado de tal manera que, ante cualquier contratiempo, tendría dos semanas más antes de la entrega del TFG.

Recursos

Recursos Humanos

En la tabla 1.5 se muestra la estimación de tiempo en horas que hice para cada uno de los apartados descritos en la planificación. Para la creación de esta tabla se tuvo en cuenta que ya estaba estipulado que las horas que debía invertir en este proyecto eran 300, por lo que traté de que la suma de la estimación fuesen las mismas.

Tarea	**Horas estimadas invertir**
Planificación	25 horas
Análisis	25 horas
Diseño	90 horas
Desarrollo	100 horas
Aprendizaje y estudio	25 horas
Seguimiento y control	10 horas
Redacción del documento	25 horas
Total	300 horas

Tabla 1.5: Recursos Humanos

Recursos materiales de desarrollo necesarios para el TFG

Para el desarrollo de este proyecto se utilizaron dos ordenadores. En primer lugar, uno portátil por la movilidad que este nos aporta para la redacción de documentos, análisis, diseño y seguimiento.

Para desarrollar el código utilicé un segundo ordenador de sobremesa por la potencia que este me aportaba.

Por último, utilicé un dispositivo móvil para poder realizar pruebas de la aplicación *Android* utilizando el ordenador de sobremesa también como servidor.

Capítulo 2

Análisis

En este capítulo vamos a ver el análisis que realicé para conocer los requisitos exactos que debía cumplir la aplicación, esto es, conocer con precisión las funcionalidades de las que tiene que disponer, y también, las tecnologías a utilizar y el motivo por el que fueron elegidas.

Para ello, primero identifiqué los requisitos que debía cumplir el proyecto. Este proyecto tiene dos partes, la parte servidora y la alojada en el dispositivo móvil, que actuará como cliente.

Tras identificar los requisitos, seleccioné las tecnologías que iba a utilizar para su desarrollo, razonando cada decisión. Finalmente, en base a los requisitos identificados, se detallaron las funcionalidades que debía tener la aplicación.

2.1 Requisitos del proyecto

La aplicación está pensada para su uso desde un dispositivo móvil en el interior de ambulancias. Una vez identificado el usuario y asignada la ambulancia, envía una petición al servidor para saber los traslados que debe realizar, junto a qué pacientes debe recoger con sus datos, como por ejemplo, la dirección. Estos datos se almacenarán de forma provisional dentro del dispositivo móvil de forma que, si el dispositivo se queda sin internet, no pierda funcionalidad.

La aplicación identificará a los pacientes que se van recogiendo, sacando una foto de su TS (también se dará la opción de escribir el número manualmente). Cada cierto tiempo, la aplicación enviará su posición al servidor para que se sepa por dónde y en qué situación se encuentra la ambulancia.

Tras hablar con Javier Virto, identificamos los requisitos mínimos que debía cumplir la aplicación:

- Desde la aplicación móvil cada usuario podrá identificarse con su nombre de usuario y su contraseña. Una vez autentificado en la aplicación, se podrá conocer

el número[1] y matrícula de la ambulancia que se le ha asignado.

- Se podrá consultar el listado de *traslados* pendientes de la unidad. De estos, se muestra: la hora estimada de finalización, estado en el que se encuentra el *traslado* (*sin comenzar*, *comenzado* o *finalizado*), si el *traslado* es para trasladar pacientes de domicilio al hospital o del hospital al domicilio, y todas las *peticiones* que conforman el *traslado*. Las *peticiones* son subtareas del *traslado*, por lo que estas contienen la siguiente información: qué médico ha solicitado el traslado del paciente, a qué parte del hospital hay que llevarlo o recogerlo, el servicio que ha de realizar el paciente, la información del propio paciente (como por ejemplo el nombre o la dirección) y el estado de la *petición* que indicará si esta se ha completado.

- Desde los *traslados* se podrá acceder a los detalles de cada una de sus *peticiones*.

- Desde la aplicación se podrá actualizar el estado del *traslado* en la BD del servidor.

- Solo podrá haber un único traslado en *comenzado*. A este *traslado* lo llamaremos *traslado actual*.

- Se podrá acceder directamente al único *traslado actual*.

- Las *peticiones* se deberán actualizar identificando a pacientes con una foto de su TS o introduciendo el número de esta a mano. Solo se podrán actualizar las *peticiones* de un *traslado actual*.

- Desde la aplicación se podrá ver el mapa de la localización del destino y se podrá acceder a la ruta a realizar desde la localización actual de la ambulancia, al destino.

Además, la aplicación deberá enviar, aproximadamente cada minuto, a la BD del servidor la posición del *traslado* para visualizar dónde se encuentra la ambulancia. También, la aplicación deberá poder ser utilizada en modo *offline*, de forma que, cuando se recupere la conexión a Internet, se actualizarán los *traslados* y las *peticiones* en la BD en el caso de que haya habido cambios.

Por otro lado, en el servidor se dispondrá de una BD que especificaremos a continuación. Los servicios de este servidor se consumirán a través de una API REST, para que los dispositivos *Android* interactúen con la BD. La BD deberá almacenar la siguiente información:

- Las ambulancias de las que dispone el hospital, con su número, matrícula y localización.

[1]Todas las ambulancias tienen un número (comúnmente de 3 cifras) que se utiliza para identificarlas de manera rápida, lo que en el usuario vendría a ser el *nickname*.

- Usuarios y sus datos: nombre de usuario, *hash*[2] de la contraseña y nombre real del usuario.

- Los distintos pacientes que hay junto con sus datos. Estos se identificarán con su número de TS, y también se almacenará su nombre y dirección domiciliaria.

- Las peticiones existentes junto con sus datos.

- Los *traslado*s a realizar y el *traslado actual*, cada uno con sus peticiones, ambulancia asignada y estado en el que se encuentra el *traslado*.

2.2 Software y tecnologías seleccionadas

En este apartado veremos qué herramientas se han seleccionado para desarrollar el proyecto y el motivo de su elección. Aunque no continué en la empresa y tuve la libertad de elegir las que más me gustasen o las que más me convinieran, utilicé las que hubiera utilizado en la empresa. En principio, esta decisión fue motivada porque más adelante existía la posibilidad de que GFI integrase este proyecto en el sistema informático del hospital, y si utilizaba las mismas herramientas que ellos, podrían mantenerlo con más facilidad.

Visual Studio

Visual Studio es el IDE desarrollado por Microsoft. Los servidores de la empresa GFI se desarrollan en C#. Como es una herramienta que ya usé en las prácticas y en el periodo que estuve contratado, estoy más experimentado usando este IDE. C# actualmente tiene una gran comunidad, un buen soporte y un abanico de plugin y ayudas que facilitan el desarrollo del código.

Entity Framework

Entity Framework es un ORM (Objecto-Relational Mapping) desarrollado para .NET. Este *framework* nos va a permitir diseñar un *dominio* en C#, definir sus relaciones y, a partir de él, crear y administrar la BD. Dentro de Entity Framework hay dos patrones de programación: *From BD to code* y *From Code to BD*. Como sus nombres bien indican, *From BD to Code* crea un *dominio* a partir de una BD dada, por otro lado *From Code to BD*, es el que utilicé para crear una BD desde unas entidades de *dominio* dadas.

[2]Función que como entrada tiene una cadena de texto y como salida un número sin ninguna relación aparente. Es casi imposible saber cuál es la preimagen de este número ya que la función no tiene inversa. Muy útil para acreditar a alguien por contraseña sin almacenarla.

Android Studio

Desde el principio, el cliente, Javier, indicó que la aplicación que se desarrollaba era para dispositivos *Android*, sin planes de futuras migraciones a otros sistemas operativos. Para ello, creí conveniente usar el IDE diseñado explícitamente para desarrollar aplicaciones en *Android*.

MySql

El SGBD elegido es MySQL. Es el SGBD que más utilicé durante mi estancia en la empresa, y como no hay requisitos en la BD que nos favorezcan usar uno u otro, elegí este.

SQLite

Es el SGBD que viene de manera predefinida en *Android Studio*.

Google Vision (OCR)

Se utiliza *Google Visions* para poder transformar de manera sencilla la imagen captada por la cámara por caracteres, y de esta manera, ser capaz de identificar a alguien a través de su TS. Se ha elegido *Google Visions* porque es el desarrollado por *Google*, empresa dueña de *Android*. En la elección se ha tenido en cuenta que *Google Visions* es el mejor, ya que está preparado para usarse en *Android* y proyectos de *Google*, sin embargo, a partir de 1000 usos mensuales deja de ser gratuito[3]. Por esto, aunque este sea barato, puede que en un futuro se prefiera utilizar algún otro servicio gratuito. Antes de seleccionar este, consideré utilizar *OCRopus*[4] o *Tesseract*[5], ambos con licencia *Apache*. Estas sugerencias pueden ser la alternativa si se decide cambiar la tecnología para la lectura de la TS.

Google Maps

He seleccionado los mapas digitales y el sistema de navegación de rutas desarrollado por *Google* para mostrar la ruta que debe seguir la ambulancia. Como su desarrollador es *Google*, tiene muchas facilidades para instalarse o referenciarse desde una aplicación *Android*.

API REST

Para la comunicación entre el dispositivo y el servidor hará uso de una API REST. A parte de que para todos los proyectos cliente-servidor en la empresa GFI, se utilizaba API REST, me pareció más adecuado que otras alternativas como SOAP o similares.

[3]Página de precios de *Google Cloud*: `https://cloud.google.com/vision/pricing`
[4]Introducción a *OCROpus* y su código: `https://www.linuxlinks.com/ocropus/`
[5]Proyecto Github de *Tesseract*: `https://github.com/tesseract-ocr/`

IIS

El servicio se despliega a través de IIS (Internet Information Services). Este servicio será instalado en el ordenador que hace de servidor, y con la IP de este, se puede usar la API REST que inicialmente diseñe.

2.3 Casos de Uso

En este apartado veremos los casos de uso que puede llevar a cabo un *usuario* de la aplicación, junto con la descripción de cada uno de estos. En este caso, solo tendremos un actor que será el *usuario*, esto es, el tripulante de la ambulancia.

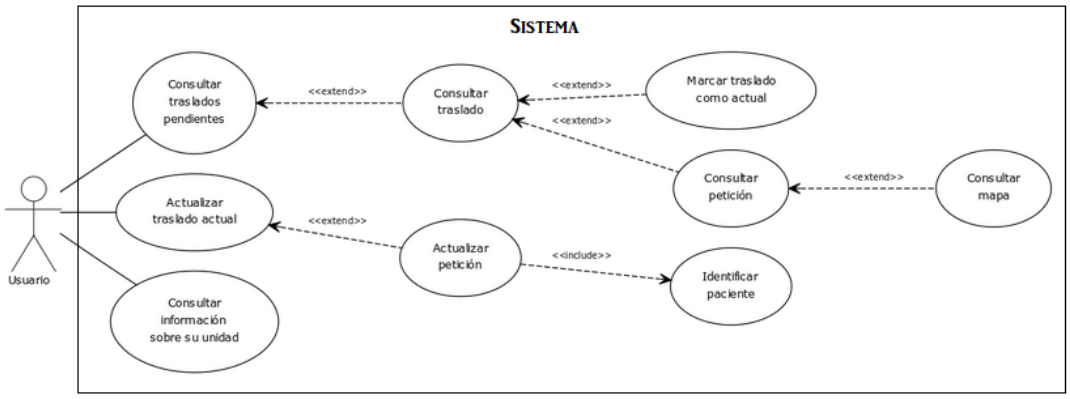


Figura 2.1: Diagrama de casos de uso

El usuario debe estar logueado para realizar cualquier caso de uso indicado en la figura 2.1.

Nombre: *Consultar traslados pendientes.*
Actores: Usuario.
Descripción: Consulta los *traslados* asignados a la ambulancia mostrándole algunos detalles: dirección a la que debe ir el *traslado* (hospital o domicilio) y la hora en la que la ambulancia debería completar el *traslado.*
Extensiones: *Consultar traslado.*

Tabla 2.1: Tabla de Caso de uso *Consultar traslados pendientes*

Nombre: *Consultar traslado.*
Actores: Usuario.
Descripción: Aparecen por pantalla la dirección del *traslado*, la hora estimada de finalización del *traslado* y una lista de las peticiones ligadas a este traslado.
Extensiones: *Marcar traslado como actual*, *Consultar petición.*

Tabla 2.2: Tabla de Caso de uso *Consultar traslado*

Nombre: *Marcar traslado como actual.*
Actores: Usuario.
Precondiciones: No puede haber ningún *traslado actual* (*comenzado* y no *finalizado*).
Descripción: Cambia un traslado pendiente a *comenzado.*

Tabla 2.3: Tabla de Caso de uso *Marcar traslado como actual*

Nombre: *Consultar petición.*
Actores: Usuario.
Descripción: Se muestra por pantalla la dirección de destino, médico solicitante y los siguientes datos del paciente: nombre y apellidos, número de la TS, y la dirección de recogida del paciente.
Extensiones: *Consultar mapa.*

Tabla 2.4: Tabla de Caso de uso de *Consultar petición*

Nombre: *Consultar mapa.*
Actores: Usuario.
Descripción: Se muestra por pantalla la posición a la que hay que dirigirse y posibilita el ver la ruta, para usar la aplicación como GPS.

Tabla 2.5: Tabla de Caso de uso de *Consultar mapa*

Nombre: *Actualizar traslado actual.*
Actores: Usuario.
Precondiciones: Todas las peticiones del *traslado actual* están completas.
Descripción: Actualiza el estado del *traslado* de *comenzado* a *finalizado.*
Extensiones: *Actualizar petición.*

Tabla 2.6: Tabla de Caso de uso *Actualizar traslado actual*

Nombre: *Actualizar petición*
Actores: Usuario.
Precondiciones: Debe ser la petición del *Traslado actual* y el *paciente* debe de estar identificado.
Descripción: Se actualiza el estado de una *petición* a *completa*.
Extensiones: *Identificar paciente*.

Tabla 2.7: Tabla de Caso de uso *Actualizar petición*

Nombre: *Identificar paciente*
Actores: Usuario.
Descripción: Bien introduciendo manualmente el número de la TS o con una foto de la misma, identificamos al paciente para actualizar la *petición*.

Tabla 2.8: Tabla de Caso de uso *Identificar paciente*

Nombre: *Consultar información sobre la unidad.*
Actores: Usuario.
Descripción: Muestra por pantalla los siguientes datos de la ambulancia: número de ambulancia y matrícula.

Tabla 2.9: Tabla de Caso de uso *Consultar información sobre la unidad*

Capítulo 3

Diseño

En este capítulo vamos a ver el diseño que se ha realizado para la aplicación, distinguiendo lo que iría en el servidor y lo que iría en el cliente. Al comenzar el diseño, le di más importancia a la parte servidor del proyecto, la que lleva consigo la *persistencia*, *dominio*, *lógica* y *WebAPI*. Más tarde, le di la importancia a la parte del dispositivo móvil. Esta representa la capa de *presentación* del proyecto aunque incluye *dominio*, *lógica* y una *persistencia* dividida en dos fragmentos. Estas capas del cliente son más ligeras que las del servidor, con el propósito de poder almacenar la información justa y necesaria para el correcto funcionamiento. Durante el proyecto nos referiremos como aplicación *Android* al cliente de este modelo cliente-servidor, que va alojado en un dispositivo móvil.

La arquitectura seleccionada para el proyecto ha sido la arquitectura de n-capas basada en *dominio*. De estas n-capas, 4 se encuentran en el servidor: *persistencia*, *dominio*, *lógica*, y *WebAPI* o *aplicación Web*. El cliente por su parte tiene las capas: *presentación*, *lógica*, *dominio* y *persistencia*. La razón por la que seleccioné esta arquitectura ha sido la misma por la cual seleccioné algunas tecnologías en el capítulo anterior: si en un momento dado GFI, integra este proyecto, será de gran ayuda que esté desarrollado con las tecnologías y arquitecturas que ellos están acostumbrados a utilizar.

Primero, diseñé la parte servidor, empezando por la capa de *dominio* donde iban las entidades. La capa de *persistencia* no habría necesidad de diseñarla, ya que con el *dominio* diseñado y desarrollado, Entity Framework crearía y gestionaría la BD.

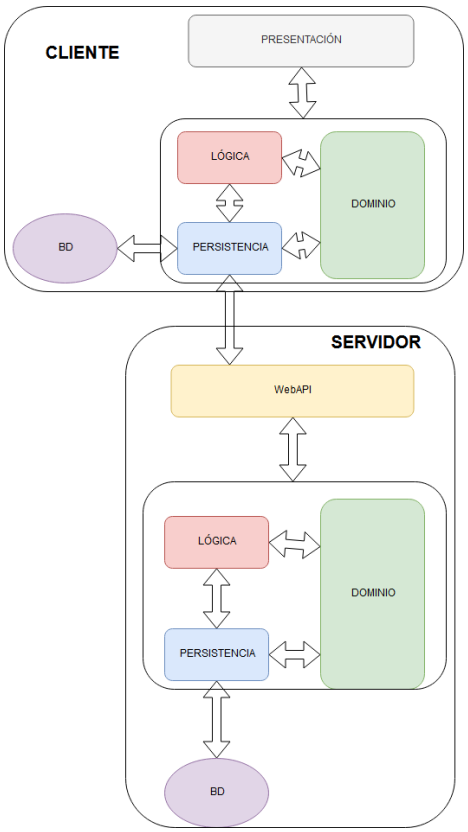


Figura 3.1: Arquitectura de n-capas basada en *dominio*

La capa de *lógica* contiene los servicios relacionados con la lógica de negocio. Por último, la capa de *aplicación Web*, define los métodos que estarán disponibles de manera remota, para que sean consumidos por el cliente. Para definir estos métodos se hará uso de una API REST.

Finalmente, he diseñado la parte cliente, que consiste en las capas de *dominio*, *lógica* y *persistencia* que necesitaré para el correcto funcionamiento de la aplicación *Android*, y la capa más relevante en el cliente, la capa de *presentación*.

3.1 Arquitectura n-capas basada en *dominio*

Esta arquitectura es la más utilizada y recomendada para el diseño y desarrollo de los microservicios. El mayor beneficio que representa esta arquitectura respecto a la de 3 capas, principalmente es una mayor independencia entre fases. Por ejemplo, en el caso de microservicios disponemos de las capas principales (*persistencia*, *dominio* y *lógica*) y además podemos disponer de las capas de *aplicaciones web*, conexión con otros microservicios, etc. Esta separación nos permite cambiar de tecnología o proveedor de cada cometido de manera más sencilla, ya que como se ha insistido, cada capa es independiente.

Para la aplicación a desarrollar vi que era necesario separar en 5 capas, 3 de ellas estarían tanto en el cliente como en el servidor, pero serían distintas. Las capas alojadas en el servidor serán las siguientes 4:

- *Persistencia:* Es la capa que, como su nombre indica, está orientada a que los datos persistan y no se pierdan. Comúnmente es la capa que conecta el programa con un SGBD para que se almacenen los datos en una BD.
- *Dominio:* Es la capa principal del proyecto, la que define qué entidades hay y cómo son. Todo el proyecto se desarrolla alrededor de respetar el comportamiento que deban llevar a cabo estas entidades.
- *Lógica:* Esta capa incluye los servicios que ofrecerá el proyecto internamente, esto es, interactúa con la *persistencia* para la obtención de datos y con el *dominio* para reflejar el comportamiento definido por este último.
- *WebAPI* o *Aplicación Web:* Esta última capa define los servicios que se expondrán al exterior, esto es, en esta capa se verán reflejados los métodos a los que se podrán acceder desde la web. Esta capa está directamente ligada a la *capa lógica*, ya que usualmente cada método de la *WebAPI* consumirá un servicio de esta capa.

Por otro lado, las capas alojadas en el cliente serán: *persistencia*, *dominio*, *lógica* y *presentación*. En este caso, la *persistencia* la he dividido en dos partes distintas: la *persistencia web* que se encargará de gestionar obtener y actualizar los datos consumiendo la API REST del servidor, y la *persistencia local* que se encargará de

gestionar la BD local para el funcionamiento de la aplicación *offline*. La capa de *lógica* transporta los datos desde la *persistencia* hasta la *presentación* gestionando las dos partes de la *persistencia*. También se encarga de actualizar ambas partes de la *persistencia* en cuanto haya acceso a Red, esto es, envía al servidor los cambios pendientes producidos en la BD local, y consulta si hay más *traslados* por realizar en la BD del servidor.

Por último, tenemos la capa más característica del cliente, la capa de *presentación*. Esta será la que interactúe con el usuario, transformando sus acciones en peticiones al servidor o a servicios del propio dispositivo.

3.2 Diseño de la capa *dominio* del servidor

Para la capa *dominio* del servidor diseñé un diagrama de clases UML, que puede verse en la figura 3.2. En este vienen definidas las entidades que necesitamos para el dominio del servidor.

La clase `Usuario` representa al portador de la aplicación. De él se almacenan su `Guid`, el atributo `Usuario` (el *nickname*), su nombre y el `hash` de su contraseña. Tiene un método que comprueba la contraseña: se calcula el *hash* de esta y se observa si son iguales. De esta manera, no se almacena la propia contraseña.

La clase `Token` no tiene más que un identificador, una fecha de expiración (24 horas desde su creación) y finalmente, contiene la identidad del usuario. Se crea un `Token` cada vez que un `Usuario` inicia sesión. Cuando ha pasado 1 mes desde la expiración del `Token`, este se borra de la BD automáticamente.

La `Ambulancia` por su parte lleva un identificador, su número de ambulancia, la matrícula de este, dónde se encuentra y los usuarios que se encuentran en la unidad (posibles portadores de la aplicación).

El `Traslado` contiene un identificador, un estado de tipo `Estado`, la ambulancia involucrada, una o varias peticiones. Además dispone de un atributo de tipo `boolean`, llamado `DireccionAHospital`, que indicará si el traslado es hacia el hospital (si es `true`) o si es hacia el domicilio (si es `false`), y por último, también se almacena la hora a la que debería finalizarse el *traslado*.

La `Peticion`, además del identificador, también lleva: el servicio hospitalario, el médico involucrado, la dirección del hospital a la que se debe enviar al paciente y los datos del propio paciente. La `Peticion` también incluye un atributo `boolean Estado`. Si el `Estado` es `false`, significa que no está completa la subtarea o `Peticion`, esto es, en el caso de que el *traslado* sea hacia el hospital significa que aún está en su domicilio y, en caso de que el *traslado* sea a domicilio, significaría que está en el hospital o en la ambulancia, dependiendo del `Estado` del `Traslado`. Si por el contrario el `Estado` de la `Peticion`, es `true`, si el *traslado* es a domicilio, significa que ya se encuentra

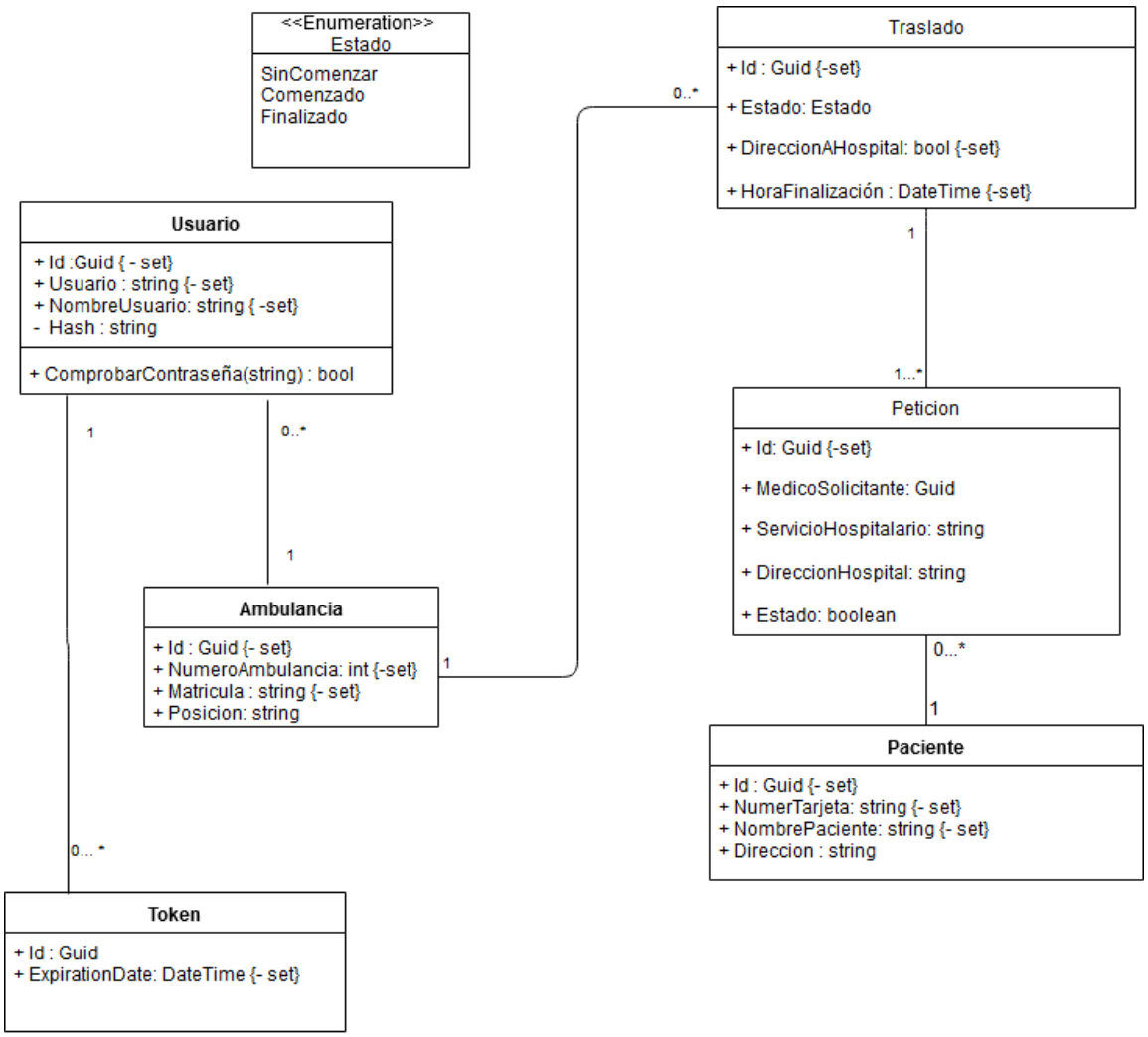


Figura 3.2: Diagrama de clases

en su hogar y si el `Traslado` es al hospital estará en la ambulancia o en el hospital, dependiendo del `Estado` del `Traslado`.

Para finalizar, el `Paciente` cuenta con: un identificador, su número de la TS, su nombre, y la dirección donde vive o hay que ir a buscarlo.

3.3 Diseño de la capa *lógica* del servidor

En este apartado veremos el diseño de las interfaces que conforman la capa de la *lógica*. Esta capa nos permite que la capa de *aplicación Web* tenga datos procesados en vez de obtenerlos directamente desde la *persistencia*. Esto hace que las capas de *presentación* o *aplicación Web* interactúen con la BD de manera "más amigable".

Las interfaces que se han desarrollado para el proyecto son las siguientes: la interfaz de servicios relacionados con el usuario, la de las ambulancias, la de los traslados, la de las peticiones y, para finalizar, la de los pacientes, es decir, una interfaz de servicios para cada entidad de la aplicación, exceptuando `Token`, los servicios de este están integradas en la interfaz de `Usuario`.

También veremos que, para poder mantener la independencia entre capas en esta arquitectura debemos crear Data Transfer Objects (Dto's), los cuales definiremos a continuación.

Dto

Un Dto es un objeto de solo lectura y escritura, serializable. Son utilizados para la transferencia de objetos entre capas o dispositivos. Solo contienen atributos, sin métodos, similares a los `struct` de `C`.

Como he comentado, antes de definir las interfaces hay que aclarar que utilizaremos Dto's para la transferencia de objetos entre el dispositivo móvil y el servidor. Estos son idénticos a los definidos anteriormente en la capa de *dominio*, solo que no tienen ningún método. Tampoco contienen información confidencial, como el `hash` de la contraseña en el `Usuario`. También se suprime información innecesaria, por ejemplo, para el envío de `Token` no se enviará todo el objeto, solo el `Guid`, ya que no hace falta nada más. En este caso particular este `Guid` viajará en el Dto de `Usuario` como un atributo más.

Cada *`NombreDeLaClase`* tiene su Dto asociado llamado *`NombreDeLaClaseDto`*, esto es, para transportar un `Traslado` tendremos el Dto `TrasladoDto`. Cada clase normalmente tiene un único Dto, pero dependiendo de las necesidades pueden tener más (o no tener). El número de Dtos vendrá dado por las necesidades que haya de recibir o enviar de distintas maneras la misma clase. Por ejemplo, la clase `Token` no tiene Dto, pero la clase *Usuario*, tiene tres, ya que tiene tres maneras de transportar un `Usuario`. Por ejemplo, uno de los Dto de `Usuario` contiene al `Usuario` completo con su contraseña en vez de `Hash`, el cual solo se utiliza cuando se desea incluirlo en la BD. El segundo Dto, el más utilizado, no incluye contraseña[1]. Por último, `UserCredentialsDto` solo transportará usuario y contraseña para el Log-in.

Mapeadores

Los Dto's son clases ajenas al *dominio*, por lo que la conversión de Dto a clase y de clase a Dto no es directa. Esto provoca que tengamos que crear unos mapeadores, es decir, unas clases con funciones estáticas para que realicen estas conversiones. En particular, nosotros generaremos una clase mapeadora por cada interfaz, y cada Dto tendrá su método de conversión de instancia de clase de *dominio* a Dto y al revés.

[1] Cuando alguien se acredite, el Dto que se le devolverá además incluirá el Token.

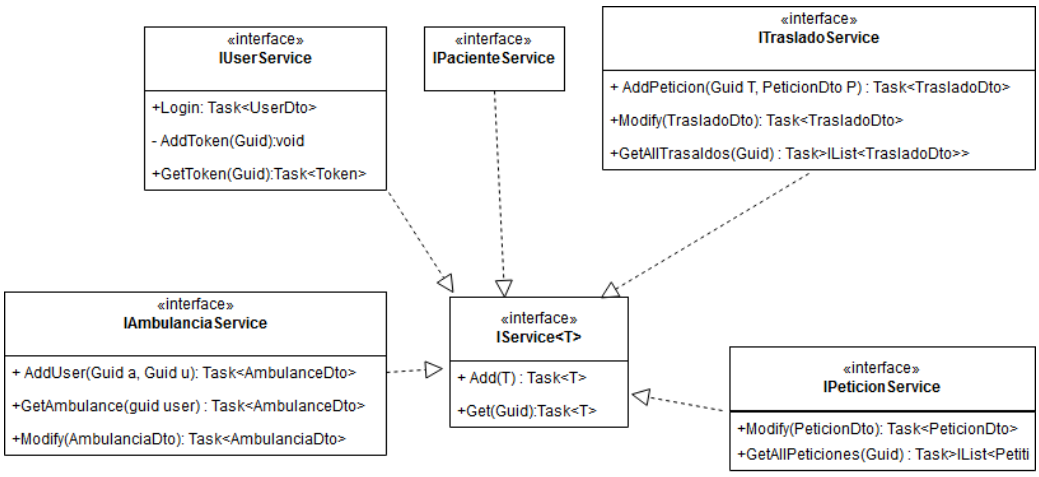


Figura 3.3: Diagrama de la capa *lógica*

Estructura de las interfaces de servicios

Cada una de las entidades principales (`Usuario, Ambulancia, Traslado, Peticion` y `Paciente`) tiene su servicio en la capa de *lógica*. Cada uno de los servicios tiene una interfaz llamada *`INombreDeLaClase`*`Service` que tiene la definición de cada método, con su descripción, precondiciones, etc. Al lado de cada interfaz se encuentra la clase que lo implementa, *`NombreDeLaClase`*`Service`, en la que se implementarán los métodos definidos en la interfaz. Todos los métodos de las interfaces tendrán como objetos de entrada y salida Dto's o tipos primitivos.

Tal y como se representa en la figura 3.3, cada uno de estos servicios tendrá 2 métodos principales: añadir a la BD y obtener un objeto de la BD. Tres clases disponen de los métodos de modificar, pero solo se podrán modificar algunos atributos. Aparte de estos tres métodos, alguna clase puede que tenga algún servicio particular, como veremos a continuación.

- `IUsuarioService:` Tiene los métodos de añadir y obtener. No tendrá modificar ya que no tiene ningún campo modificable. Para el uso del método particular, `Login` se utilizará otro servicio particular, `AddToken` ya que, dados un nombre de usuario y contraseña comprueba en la base de datos si existe ese `Usuario` y si esa es su contraseña. Después, creará un `Token` de sesión y esto le dará acceso al resto de servicios. Por otro lado, el otro servicio será el de obtener un `Token` dado su `Guid`.

- `IAmbulanciaService:` Como en `Usuario`, tenemos los métodos de añadir y obtener, pero esta vez también tenemos un método de modificación (el único atributo modificable será el de `Posicion`). Como servicio particular tenemos al usuario que se le asigna una ambulancia y la obtención de una ambulancia a partir del `Guid` de un `Usuario`.

- `ITrasladoService`: En estos servicios también tenemos los tres métodos antes descritos. En este servicio solo podremos modificar el valor del `Estado` del `Traslado`. Como servicio particular tendremos también dos, añadir una `Peticion` al `Traslado` y obtener todos los `Traslados` de una ambulancia.

- `IPeticionService`: En estos servicios también encontramos los 3 métodos principales, permitiendo únicamente modificar el `Estado` de la `Peticion`. Como servicio particular, disponemos de obtener todas las peticiones de un mismo traslado.

- `IPacienteService`: Por último, tenemos los servicios orientados a `Paciente`. De este solo se puede obtener y añadir. No permite modificación.

Los `Token`, por su parte, no dispondrán de servicios. Tanto añadir un `Token` como obtener un `Token` estarán integrados en `IUsuarioService`.

3.4 Diseño de la capa *aplicación Web*

El diseño de la *WebAPI* está orientado exclusivamente al funcionamiento correcto de la parte cliente. Los métodos de esta capa acceden directamente a algunos de los métodos de la capa *lógica*, por lo que tendrán el mismo nombre que los servicios utilizados, como se ve en la figura 3.4. En esta figura podemos echar en falta los servicios del *paciente*, pero cada vez que recibamos *peticiones*, recibiremos el Dto del *paciente* en el interior del Dto de la `Peticion`. Por razones de seguridad, actualmente, servicios como añadir elementos a la BD, no estarán disponibles desde la *WebAPI*. Esta API REST, nos permitirá ofrecer los servicios del servidor a dispositivos externos. Estos dispositivos podrán consumir el servicio accediendo a la URL que se fije para dicho servicio a través del protocolo HTTP. Para la mayoría de estos servicios habrá que añadir los recursos de entrada en la cabecera, cuerpo o en la propia ruta. Las respuestas serán códigos HTTP con el contenido correspondiente que tenga que devolver.

Los Dto que se transporten a través de esta API REST, irán en formato `json`.

Excepto el login, el resto de servicios necesitan que vaya el `Guid` del `Token` en la cabecera de la petición REST. Los códigos de respuesta que se podrán recibir serán los estándar:

- `200`: Si todo ha sido correcto, el 200 irá acompañado del recurso solicitado.

- `403`: Este código se envía cuando no se está autorizado para recibir el recurso, esto es, cuando el `Token` no es correcto o el login no ha ido bien. Nunca irá acompañado por más que el código.

- `404`: Se enviará cuando no se encuentre el recurso solicitado. Si el recurso no existe y no estás autorizado para recibirlo recibirás un `403`.

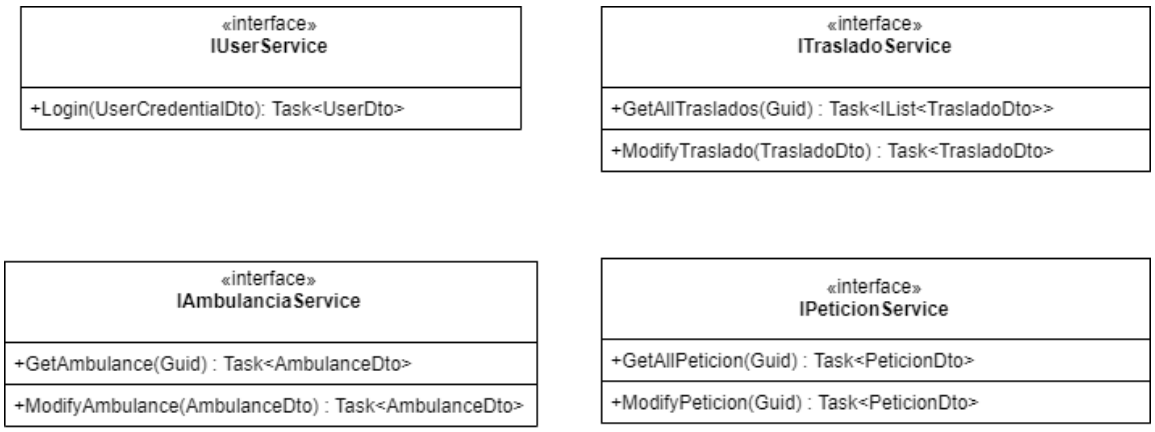


Figura 3.4: Diagrama de la capa *aplicación Web*

Las URL que utilizaremos para los servicios serán `direccionServidor/api/X/Y` siendo `X` el nombre del servicio (`ambulancia, usuario, traslado` o `peticion`) e `Y` será el servicio que se solicita.

Los servicios que dispondremos serán:

- POST `api/usuario/login`: Sirve para loguearse. Devuelve el `Usuario` que se ha logueado o un `Unauthorized`.
- GET `api/ambulancia/usuario/{id}`: devuelve la `Ambulancia` del usuario con el `Guid` id.
- PUT `api/ambulancia/modify`: modifica la localización de la `Ambulancia`.
- GET `api/traslado/ambulancia/{id}`: devuelve todos los `Traslados` de la `Ambulancia` con ese Id.
- PUT `api/traslado/modify`: modifica el `Estado` del `Traslado`.
- GET `api/peticion/traslado/{id}`: recibe todas las `Peticiones` del `Traslado` con esa id.
- PUT `api/peticion/modify/{id}`: cambia el `Estado` de una `Peticion` a `True`.

Los métodos `modify` toman como referencia el `Id` del Dto de entrada para identificar el elemento a modificar. Posteriormente modifica todos los campos modificables, al estado en el que se encuentran en el Dto de entrada.

3.5 Pruebas del servidor

Las pruebas unitarias del servidor se separarán en tres fases: las referentes a *persistencia*, *dominio* y *lógica*. Las pruebas de la *WebAPI* se realizarán manualmente con

la herramienta *PostMan*[2].

Para las pruebas de la *persistencia* se tratará de añadir, eliminar, obtener y modificar un recurso de cada una de las entidades, comprobando que el comportamiento del SGBD es correcto.

Para la parte de *dominio*, para cada clase se hará una prueba donde se creará una instancia de esa clase, y se probará que los datos de creación son correctos, que se modifica correctamente y si dispone de algún método de prueba.

Por último, para probar la capa *lógica* crearemos *Moq*s para simular el comportamiento de la capa de *persistencia*. Gracias a estos podremos probar todos los métodos de esta capa independientemente de los resultados de la prueba de *persistencia*.

3.6 Diseño de la capa de *dominio* de la aplicación móvil

Como podemos ver en la figura 3.5, el *dominio* que dispone la aplicación móvil es muy similar a la del servidor. La gran diferencia radica en que el acceso de los atributos no son `public`. Todas las clases tendrán los get y set típicos de *Java*, que no aparecen en el diagrama. Ninguna clase tendrá ningún método que no sea ni getter ni setter. Cada una de estas clases podríamos equipararla con un `struct` de C o un Dto del servidor.

Notar que no hay clase `Token`: el `Usuario` tiene un atributo que transportará el `Guid` del `Token` de la sesión actual. Por último, notar que solo puede haber un único `Usuario` y una única `Ambulancia`, ya que no tiene sentido que en la aplicación móvil haya más.

3.7 Diseño de la capa *persistencia* de la aplicación móvil

Como he indicado anteriormente, la *persistencia* del cliente tiene dos partes, las llamaremos *persistencia web* y *persistencia local*. La primera de ellas accede a los datos que se encuentran en el servidor. A estos se accederá desde la API REST que se ha diseñado para el servidor.

La segunda *persistencia* está orientada al modo offline. La capa *lógica* se encargará en cada momento de cuál de las dos *persistencia*s utilizar, por lo que veremos más adelante cuándo se activan, pudiendo activarse únicamente una de ellas o ambas a la vez.

[2]Página web de PostMan: `https://www.getpostman.com/`

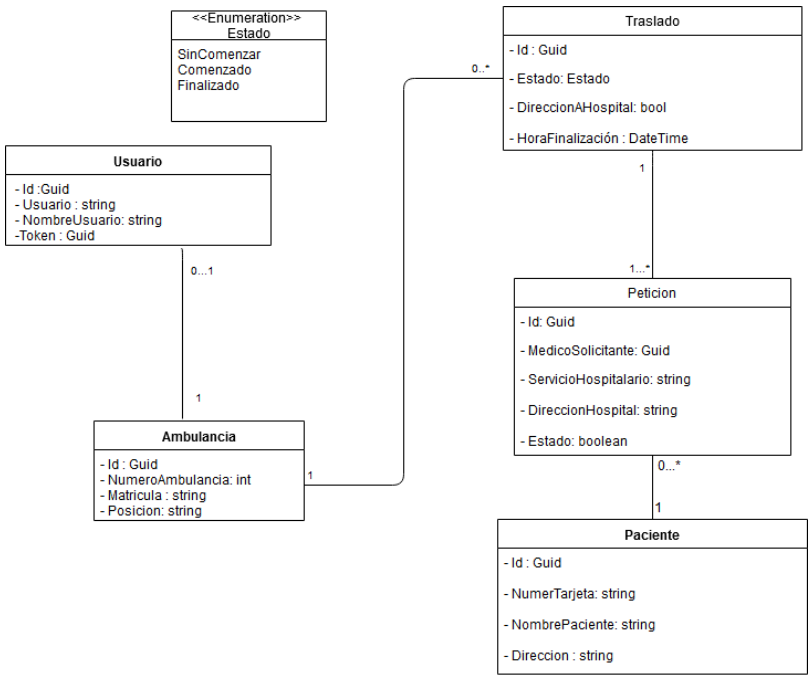


Figura 3.5: Diagrama de la capa *dominio* del cliente

Esta segunda aprovechará que *Android* tiene incorporado de manera predefinida el SGBD *SQLite*.

Persistencia Web

La *persistencia Web* hará uso de la *WebAPI* del servidor, en particular, será la encargada de consumir la API REST, por lo que dispondrá de exactamente los mismos métodos que la figura 3.4 de la sección 3.4.

Persistencia local

A continuación, veremos los métodos que dispondremos para acceder a la BD local. Esta BD solo contendrá *Traslado*, *Peticion* y *Paciente*. Al solo existir un único *Usuario* y una única *Ambulancia*, estas últimas clases las almacenaremos en una hoja de preferencias de *Android*, más accesibles que la BD.

Las hojas de preferencias son documentos comunes para toda la aplicación *Android*. Se diseñaron para guardar preferencias, como por ejemplo el tamaño de la letra que se desea. De esta manera, toda la aplicación tiene acceso a esa configuración. Actualmente se utilizan también para almacenar todos los recursos que son comunes para todo el programa, en nuestro caso, el `Usuario`, la `Ambulancia` o el `Token`.

Los métodos que tendrá esta *persistencia* serán los que mostraremos a continuación.

Para el siguiente método, `X` puede ser sustituido por: `Ambulancia`, `Traslado` y `Peticion`. De `Ambulancia` se podrá cambiar la localización, de `Traslado`, su estado o el de una `Peticion` de ese `Traslado` y asimismo, de `Peticion`, solo podremos modificar el estado. La variable modificable se cambiará a la que se encuentra en el `XDto` de entrada. Si en la BD u hojas de preferencias, no hay ningún `X` con la id idéntica a la de *`X`*`Dto`, mostrará un error a través de la pantalla.

```
public void modifyLocalX(XDto _in);
```

A continuación, veremos las funciones para obtener los datos de la BD o de las hojas de preferencias. La *`Y`* puede ser sustituida por: `Usuario`, `Ambulancia`, `Traslado`, `Peticion` o `Paciente`. Recordemos que en esta BD del *Token* solo se almacena el id y está incluida en *Usuario*. Si en la BD no hay ninguna *`Y`* con la id _id, mostrará un error por pantalla y devolverá `null`.

```
public YDto getLocalY(string _id);
```

Los siguientes métodos eliminan filas de la BD local cuando estas ya no son necesarias. La *`Z`* puede ser sustituida por: `Traslado`, `Petición` y `Paciente`. Este método elimina de la tabla *`Z`* la fila que tenga la id _id, si no existe, mostrará un error por pantalla.

public void deleteLocal*Z*(string _id);

Esta última función sirve para eliminar todos los datos de la BD y de las hojas de preferencias. Solo se activa cuando se sale de la aplicación y cuando el usuario se dispone a loguearse.

```
public void deleteAll();
```

3.8 Diseño de la capa *lógica* de la aplicación móvil

Esta capa hará de intermediaria entre ambas partes de la *persistencia* y la *presentación*. Los métodos que dispone son los mismos que dispone la *WebAPI* del servidor y alguno más que especificaremos más tarde. Cada vez que hagamos cualquier operación esta capa registrará los datos en la BD del móvil a través de la *persistencia local*, y cuando se pueda se enviarán al servidor a través de la *persistencia Web*.

Cada vez que recibimos cualquier dato desde el servidor, esta capa se encargará de almacenarlo desde la *persistencia local*, bien sea en la BD local o en la hojas de

preferencias (en el caso de `Usuario` o `Ambulancia`). Si al tratar de obtener información no tenemos conexión a internet, informará de ello y devolverá los datos que se encuentran almacenados en la BD local.

Cada vez que se haga cualquier operación que requiera de actualizar cualquier objeto, esta capa tratará de enviársela al servidor y también grabarlo en la BD local. Si este primero no es accesible en la hojas de preferencias marcará un *flag* de actualización pendiente y si se ha cambiado de `Traslado` (porque se ha marcado como *Finalizado*), también se almacenara el `Guid` de este. En cuanto disponga de internet, si este *flag* está activo enviará los datos pendientes de enviar del `Traslado` actual. Si ha habido un cambio de `Traslado`, antes de actualizar el `Traslado actual`, enviará los del pendiente de actualizar.

Cada vez que se cierre la aplicación de forma estándar (no forzando cierre) y cada vez que la iniciemos, borrará por completo la BD y hojas de preferencias local.

Finalmente, esta capa se encarga de procesar los datos que nos devolverá *Google visions* (el OCR) para actualizar una `Peticion` y de solicitar los mapas de *Google Maps* cuando el usuario requiera la posición a la que deben de dirigirse. Si el usuario desea ver la ruta y usar el GPS, se le redireccionará a la aplicación *Google Maps*, donde encontrará la ruta ya preparada y con el dispositivo listo para hacer de GPS.

3.9 Diseño de la capa de *presentación*

A continuación, se mostrarán los prototipos de interfaces que se han diseñado para la aplicación: el aspecto que tiene cada una y las funcionalidades de las que dispone.

Pantalla de Login

En la figura 3.6 se puede ver que la parte superior de la pantalla está ocupada por el logo. En el caso de que este proyecto acabase en uso en la comunidad de La Rioja, este sería el de *Rioja Salud*.

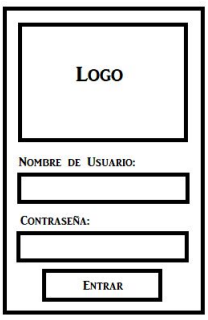


Figura 3.6: Prototipo de pantalla de *Login*

Dispone de dos entradas de texto: la primera, una entrada común para introducir el nombre de usuario. En la segunda entrada debe ir la contraseña del usuario, ya que este campo está preparado para introducir contraseñas, esto es, en vez de aparecer lo que el usuario escribe, aparecerán *-s.

Una vez introducidos los datos, pulsando en *Entrar*, se envían las credenciales al servicio web. Si todo es correcto (el servicio devuelve un código 200), se accederá a la

pantalla de *Menú Principal* (Figura 3.7). En caso contrario, el usuario se mantendrá en la pantalla actual donde se mostrará un mensaje de error de credenciales.

Menú Principal

Este menú (figura 3.7) dispone de 3 botones. Desde *Traslados Pendientes*, se puede acceder a la pantalla *Lista de Traslados*, este mostrará la lista de traslados pendientes asignados a la unidad.

Por otro lado, pulsando en *Traslado Actual*, se accede a los datos del *traslado* que la ambulancia asignada está llevando a cabo en estos momentos. También se puede actualizar la situación del traslado u acceder a las *peticiones* de este.

Por último, pulsando en *Datos de la Ambulancia*, se puede acceder a los datos de la unidad asignada al usuario. En este aparecerán su identificador, su matrícula, los usuarios que la tripulan (junto con su nombre y apellidos) y la localización de la ambulancia.

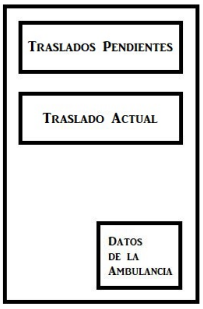


Figura 3.7: Prototipo de pantalla *Menú Principal*

Lista de Traslados

Si en el menú principal pulsamos en *Traslados pendientes*, accederemos a esta pantalla donde aparece la lista de traslados que debe realizar la ambulancia con su id y la hora a la que se debe realizar.

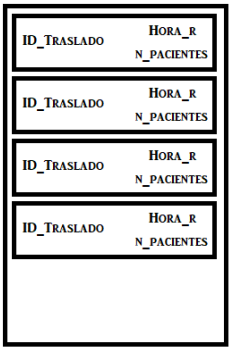


Figura 3.8: Prototipo de pantalla *lista de Traslados*

Si pulsamos en cualquiera de los traslados, nos llevará a *Traslado* donde podremos ver los detalles del traslado pulsado.

Traslado o Traslado Actual

En la pantalla de *Traslado* encontraremos los datos principales del *Traslado* o *Traslado Actual*. En la parte superior encontramos: id de la *Traslado* si el traslado es hacia el hospital o hacia el domicilio de los pacientes y hora programada para la llegada de los pacientes (que no la hora estimada calculada en tiempo real).

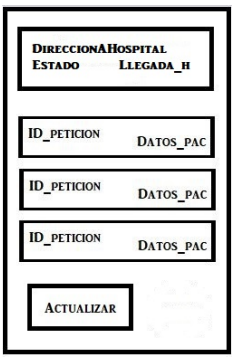


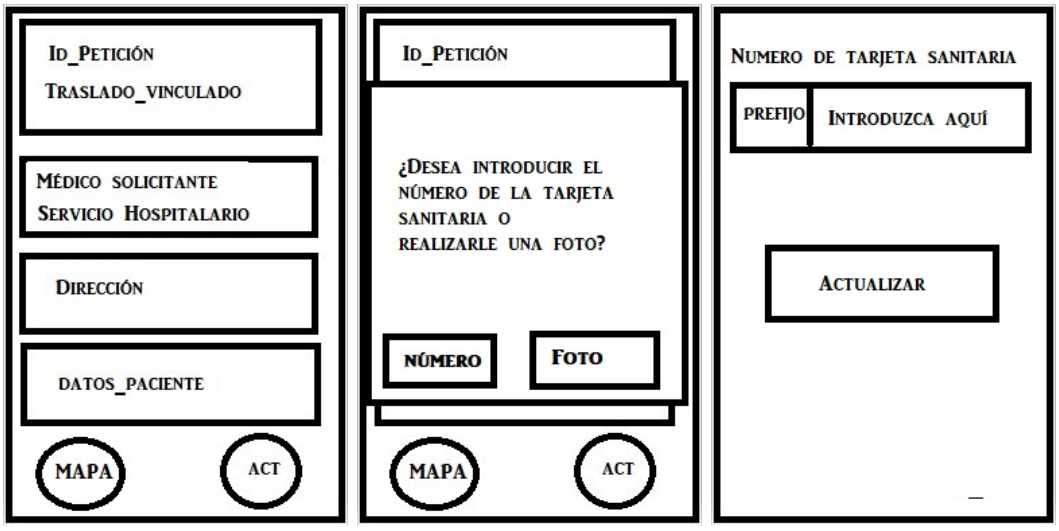
Figura 3.9: Prototipo de pantalla *Traslado*

En la parte media del formulario tenemos las peticiones que tiene el traslado. En cada una de ellas viene el id de la petición, nombre del paciente y dirección de este. Si pulsamos en una de las peticiones abriremos la pantalla *Detalles de la Petición* con datos más detallados de la *Petición*.

En la parte inferior, nos toparemos con un botón, el de actualizar, que solo aparecerá si el *Traslado* es el *Traslado Actual* o si no hay ningún *Traslado Actual*.

Detalles de la Petición

En la pantalla *Detalles de la Petición* encontramos una pantalla con poca interacción. En la cabecera de la pantalla nos aparecen el id de la *Petición* y el id del *Traslado* al que está vinculado. Por otro lado, en el cuerpo nos aparece el médico que ha solicitado al paciente junto al servicio médico que debe realizar, la dirección donde vive, nombre del paciente y número de tarjeta sanitaria.



(a) Principal (b) Dialogo (c) Introducción del número de TS

Figura 3.10: Prototipo de pantalla *Detalles de la Petición*

En la parte inferior de la pantalla encontramos dos botones. El primero sirve para actualizar el traslado. Para poder realizar esta acción deberemos utilizar la cámara para sacar una foto a la TS del *paciente* o introducir el número de esta. Si todo es correcto, cambiará el estado de la petición a *Completado*. El otro botón es para acceder al mapa, con la ubicación del destino y con opción de mostrar la ruta a través de *Google Maps*.

Capítulo 4

Desarrollo

En este capítulo voy a presentar cómo he desarrollado el proyecto en base a lo diseñado en el capítulo anterior.

Primero observaremos cómo he desarrollado la parte del servidor, cómo está estructurado y pequeñas porciones de código a modo de ejemplo de cómo se ha realizado.

Una vez finalizado el servidor, seguiremos el mismo proceso, pero esta vez con el desarrollo de la parte cliente.

4.1 Desarrollo de la parte servidor

Lo primero que desarrollé fue la capa de *dominio*. Como hemos indicado previamente, esta es la capa central del servidor por lo que, antes de desarrollar cualquier otra parte, conviene tener desarrolladas todas las entidades que participan. La organización que llevé a cabo en el *dominio* trataba de agrupar en carpetas las clases principales junto con sus añadidos, esto es, en la carpeta de `Usuario` se encontraban las clases `Usuario` y `Token`. De la misma manera, `Peticion` y `Traslado` también compartían carpeta.

Una vez desarrolladas las 6 clases, empecé trabajando con la capa de *persistencia*. Esta supuso algo más de trabajo. Aunque en la empresa utilizásemos *Entity Framework*, siempre estaba ligado a utilizarlo con librerías desarrolladas por la empresa que facilitaban las configuraciones. El primer problema que me surgió fue que el proyecto que estaba desarrollando requería de una versión previa de *Entity*. Por ello, tuve que instalar manualmente la versión compatible con *.NET Core 2.1.1*. Se escogió una versión desactualizada de *.NET* porque era la más recomendada para desarrollar Controladores API REST. Una vez instalado, descubrí que lo que sabía de *Entity Framework* era insuficiente para trabajar con él. Por lo que tuve que estudiar su funcionamiento. Una vez aprendido, mapeé todas las clases, para definir las tablas de la BD y generé el contexto de la BD (`DbContext`). El contexto de la BD es el objeto que crea las migraciones, se le añaden los mapeadores que se hayan creado y

automáticamente genera el código para crear/actualizar la BD a través del SGBD. También nos permite realizar *query*s a dicha BD. Posteriormente, creé una migración con *Entity Framework* para poder generar de manera automática la BD.

En este proyecto veremos dos tipos de mapeadores, mapeadores de *persistencia* y mapeadores de Dto. Los mapeadores de *persistencia* definen cómo son las tablas de la BD utilizando el *dominio*. Los mapeadores de Dto, como veremos más adelante, transforman Dto en instancias de la clase original o instancias de las clases en Dto.

Durante el desarrollo, me vi incapacitado de instalar el *MySql Server* tal y como pretendía. Esto hizo que terminase utilizando *Sql Server*. Por fortuna, *Entity Framework* me dio muchas más facilidades a la hora de trabajar con el SGBD de *Microsoft* que con cualquier otro SGBD.

Tras ello, creé los test de la capa *persistencia*. Estos consistían en añadir un objeto a la BD, comprobar que estuviese, eliminarlo y comprobar que no estuviese. Para realizar los test unitarios utilicé los paquetes de *FluentAssertions* que me permitirán disponer en todos los objetos las funciones `Should(), Be(Object o), NotBeNull()`, etc. Muy útiles a la hora de crear test. También usé el paquete *XUnit* para gestionar estos test. A continuación, veremos un pequeño ejemplo del tipo de test que nos ofrecen estos dos paquetes:

```
UsuarioAgg user1 = new UsuarioAgg(user, name,
pass);

user1.Should().NotBeNull();
user1.Id.Should().NotBeEmpty();
user1.NombreyApellidos.Should().Be(name);
```

En este código podemos observar cómo creamos una instancia de `Usuario`, nos aseguramos de que no sea nulo, que tenga id y que efectivamente su nombre sea `name`.

A continuación, desarrollé los test para la capa de *dominio*. Estos consistían en crear una instancia del objeto en cuestión y comprobar que todos sus atributos fueran los correctos. Más tarde, modifiqué estos atributos y comprobé que las modificaciones se habían realizado de forma correcta, tal y como hemos visto en ejemplo anterior.

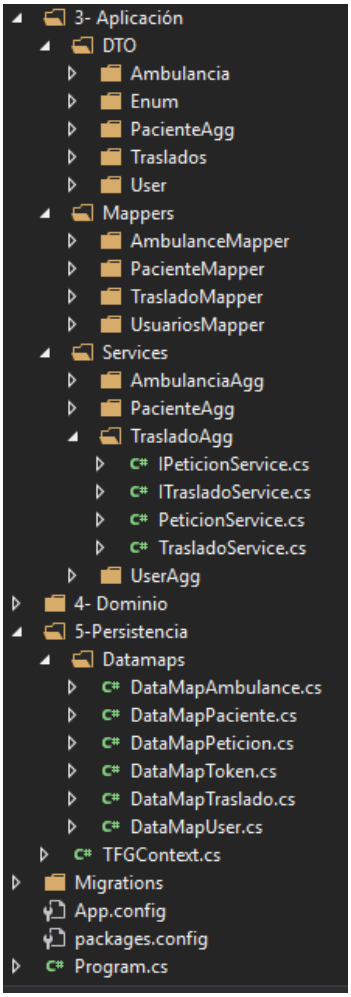


Figura 4.1: Gestión de archivos de *persistencia*, *dominio* y *lógica*

Tras comprobar que el *dominio* y la *persistencia* funcionaban correctamente, me dispuse a desarrollar la capa *lógica*. Lo primero que hice fue escribir las interfaces que debían seguir los servicios. Todos los métodos fueron documentados por el sistema de documentación predefinido de .NET, esto es:

```
/// <summary>
/// Resumen de la función.
/// </summary>
/// <param name="_param1">Definición de parámetro de entrada 1</param>
/// <param name="_param2">Definición de parámetro de entrada 2</param>
/// <returns>Descripción del parámetro/s de salida </returns>
ToReturn Método (P1 _param1, P2 _param2);
```

Este tipo de formato es muy útil para obtener documentación automática. En particular, nos ofrece la opción de redactar un resumen del método y el significado de cada parámetro de entrada/salida.

A continuación, desarrollé todos los Dto's que fuese a necesitar. Estos Dto necesarios serían todos los que hubiera como entrada/salida entre las cinco interfaces que había desarrollado previamente. Los agrupé por carpetas como aparece en la figura 4.1. No hay que olvidar que también tuve que hacer un Dto del enumerador `Estado`.

Una vez acabado el desarrollo de los Dto, comencé a desarrollar los mapeadores de Dto. Esta tarea resulta mecánica en el caso de transformar de una instancia de la clase a Dto. Para realizar esta acción, debemos obtener los atributos de la instancia de la clase a transformar y posteriormente asignar estos valores a su campo correspondiente en el Dto. Para el otro cambio, obtienes los valores del Dto y generas una instancia de la clase con su constructor. La única dificultad que me encontré en esta área fue volver al *dominio* para generar constructores de los objetos que me permitiesen tener como parámetro de entrada un `Guid` y, de esta manera, poder crear correctamente el objeto proveniente del Dto. Hasta ahora, los `Guid` de los objetos se creaban de manera aleatoria, evitando que hubiese dos objetos distintos con el mismo `Id`.

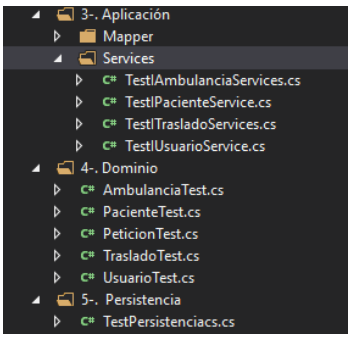


Figura 4.2: Gestión de archivos de los test

A la hora de desarrollar los mapeadores no hizo falta generar un mapeador del enumerador `Estado` de la clase `Traslado`. En la clase que este enumerador se utiliza se hace un *casting* del enumerador, en cambio, en proyectos más grandes sí se suelen utilizar mapeadores por si el orden de los enumeradores varía de una capa a otra. No obstante, en este caso, nos bastó con realizar un *casting* en las clases que lo iban a utilizar.

Finalmente, desarrollé las clases que implementan las interfaces de los servicios. Cuando se está trabajando en esta capa, constantemente se utilizan los mapeadores. Todos los parámetros que entran y salen son Dtos, pero internamente, para interactuar con el *dominio* y la *persistencia*, se trabaja con instancias de las clases del *dominio*.

A continuación, realicé los test unitarios de la capa *lógica* para probar que el comportamiento de todos los métodos fuera el correcto. Para ello, utilicé los *Moq*. Los *Moq* son objetos que simulan el comportamiento de otros objetos programándolos para que siempre devuelvan ciertos valores a ciertos parámetros de entrada. Estos son utilizados para que las pruebas sean independientes a las capas y para que un error en la capa de *persistencia* no condicione al test unitario de la capa *lógica*. Yo, por ejemplo, podía simular el comportamiento de una `DbContext` y aislar todos los errores que pueda generar esta clase, centrándonos únicamente en testear la capa aplicación.

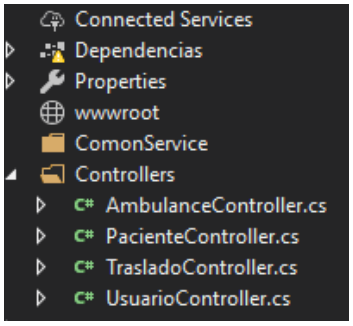


Figura 4.3: Controladores de la *aplicación Web*

Finalmente, vamos con la última capa del servidor, la *WebAPI*. Para ello creé cuatro clases que heredan de `Controller` con el prefijo de su ruta determinada en el diseño. En estos controladores tuvimos que indicar no solo cuáles eran los parámetros de entrada, sino también dónde se debían transportar (en el cuerpo, cabecera o *URI*) y a qué ruta había que referenciarse para acceder al servicio. Un ejemplo de la declaración de un servicio sería:

```
[HttpGet]
[Route("{id}")]
public async Task<IHttpActionResult> GetAmbulance([FromURI] string id)
```

Antes de la definición de este método hemos indicado que el prefijo de este servicio es `api/v1/ambulancia`, por lo que la URL para acceder a este servicio es `DireccionServidor/api/v1/ambulancia/ID` donde el valor de `ID` será el `Id` de la ambulancia que buscamos.

Para empezar, tenemos el `[HttpGet]` que indica qué tipo de petición HTTP es, de igual manera, tenemos `[HttpPost]` o `[HttpPut]`. A continuación, tenemos la ruta que se añadirá al prefijo marcado anteriormente. Y por último, tenemos el propio método.

Notar que el proyecto automáticamente se había ligado al textitIIS Express del ordenador, pero solo funcionaba en el *localhost*, por lo que haciendo uso de un paquete llamado *Conveyor*[1] pude extender el servicio a toda la red privada, claro está, no utilizando ningún *Firewall* en los puertos donde trabajaba nuestro servidor.

[1]Descarga de *Conveyor*: `https://marketplace.visualstudio.com/items?itemName=vs-publisher-1448185.ConveyorbyKeyoti`

4.2 Desarrollo de la parte cliente

En esta sección veremos el proceso de desarrollo de la parte cliente de este modelo cliente-servidor. Nada más crear el proyecto, creé una carpeta donde se alojaría el *dominio*. Como ya indiqué en el diseño, estas clases serán como los Dtos, solo con atributos, sin métodos. Esto nos permite interactuar con la *WebAPI* del servidor.

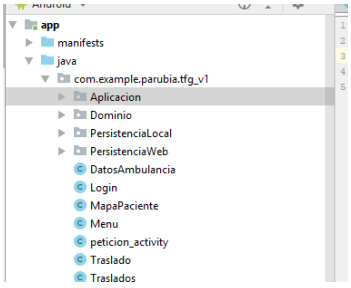


Figura 4.4: Gestión de archivos java de *Android*

Una vez creado el *dominio*, desarrollé la *persistencia local* con *SQLite*. Su desarrollo no resultó complicado. El único problema que tuve resultó ser que *SQLite* solo admite tres tipos de valores (aparte del `null`): `text, real` e `integer`. Esto hizo que tuviese que tener cuidado con tipos como el `Date`, el `Guid` o localizaciones. El `Guid` o la localización se podían almacenar de manera sencilla como un `String`, sin generar inconveniente alguno. Para esta *persistencia* creé un `DataBaseHelper`. Este *helper*, gestiona las migraciones de la BD local y facilita la inserción y extracción de datos en *SQLite*. También desarrollé varios `DataBaseAdapter`, en concreto, uno por cada tabla. Cada *adapter* gestiona la inserción, eliminación y actualización de las filas de cada tabla utilizando el `DataBaseHelper`, permitiéndonos interactuar con la BD de manera más sencilla. Veamos un ejemplo de inserción y extracción de datos dentro de un `DataBaseAdapter`:

```
//newValues funciona como un Dictionary, que es un objeto que contiene
duplas clave-valor. Así, utilizaremos como claves las columnas de cada
tabla y como valores los que se quieran introducir.

db = dbHelper.getWritableDatabase();
long result=db.insert("PACIENTE", null, newValues);

//Cursor, almacena todos los resultados de una query, en este caso
la búsqueda de todos los pacientes con el Guid igual a id.

db=dbHelper.getReadableDatabase();
Cursor cursor=db.query("PACIENTE", null, "ID=?", new String[]{id},
null,null, null);
```

Una vez creada la *persistencia local*, creé un método para insertar un `Usuario`, una `Ambulancia`, tres `Traslados`, nueve `Peticiones` y siete `Pacientes` para poder realizar pruebas de manera independiente, sin usar el servidor.

A continuación, desarrollé la capa *lógica*. Primero, creé los métodos originales que se utilizarían una vez finalizada la aplicación y más tarde, desarrollé una segunda capa *lógica* extra mientras transcurre el desarrollo. Esta segunda capa simplemente hacía de túnel entre la *presentación* y la *persistencia local* y mientras usase esta, dejaría comentada la *lógica* final.

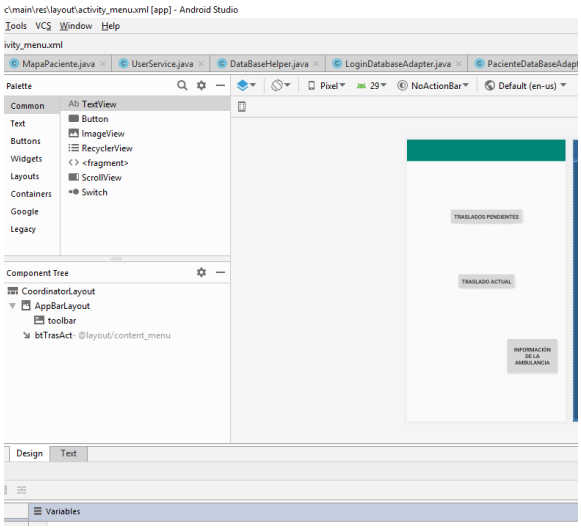


Figura 4.5: Diseño de layouts

Una vez acabada la capa *lógica*, creé todas las *actividades* de la *presentación*. En *Android Studio* las *actividades* son lo que en el *diseño* hemos llamado interfaces. Por un lado, cada una de estas *actividades* está compuesta por uno o más *xml*[2] llamados *layouts*, que representan el posicionamiento de las componentes. Por ejemplo, en estos *xml* nos aparecerá que hay un botón con cierto tamaño y que se encuentra en alguna posición en concreto. Por otro lado, está el archivo *Java* que se encarga de todas las componentes dinámicas: qué hace cada botón, qué aparece en los textos no dinámicos, etc.

Para el desarrollo de los *layout*, *Android Studio* tiene plantillas que crean automáticamente el *xml* según le indiquemos dónde y cómo se va a colocar cada componente, tal y como aparece en la figura 4.5. Esta tarea es bastante complicada, si no has trabajado previamente con *layouts* de este estilo. Todos los elementos del *layout* tienen posiciones relativas con otros elementos, y el mover uno de ellos te puede descuadrar toda la interfaz.

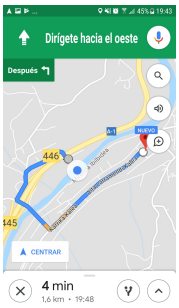
Una vez desarrollados todos los *layouts*, programé todos los ficheros *java* de la *presentación*. Al principio solo programé la aparición dinámica de `Traslados` y `Peticiones` dentro de cada `Traslado`. Más tarde, programé cada cambio de *actividad*, esto es, todos los `OnClickListener` de los botones. Los `OnClickListener` son los métodos que se disparan cuando pulsamos el botón que lo contiene, y así, se consigue que nos desplacemos de una *actividad* a otra pulsando botones.

A continuación, fui a añadir los mapas de las rutas con *Google Maps*. Aparentemente era sencillo: obtener a través de *Google* una *KEY* de *Google Maps*[3] y luego simplemente marcar un lugar e indicar cómo llegar a él. Para mi sorpresa, me encontré con un error en la generación predefinida de código en una Google Maps activity.

```
SupportMapFragment mapFragment = (SupportMapFragment)
getSupportFragmentManager().findFragmentById(R.id.map);
```

[2] Puede haber múltiples *xml*, ya que dentro de un *xml* puede haber `imports` que incluyan más archivos.

[3] Página de obtención de la KEY de *Google Maps*: `https://console.developers.google.com/flows/enableapi?apiid=maps_android_backend&keyType=CLIENT_SIDE_ANDROID&r=C9:1D:1A:30:B1:9D:C7:B3:6B:F1:05:47:D4:CB:B3:F3:20:A5:3B:CE\%3Bcom.example.parubia.tfg_v1`



Andorid Studio me daba un error porque no podía hacer el casting del fragmento de la derecha (tipo `Fragment`) en un `SupportMapFragment`. Esta línea de código es la que nos permite visualizar una zona del mapa en la pantalla. Para hacerlo funcionar debía migrar la aplicación a una versión llamada *AndroidX*. Una vez migrado me daba un error mayor, puesto que no era capaz de interpretar los *layout* que había programado previamente. Este problema se produjo porque el proyecto no se había migrado correctamente a la versión deseada, por lo que fui cambiando uno a uno todos los *layout* anteriormente programado a una versión superior. Todos los cambios a realizar se encuentran en la siguiente página web[4]. Una vez migrado a *AndroidX* todos los *layout* nuevos venían migrados, por lo que no había ningún problema con ellos.

Una vez solucionado el problema, creé un método para calcular coordenadas a través de direcciones (transformar de `Hospital San Pedro, La Rioja` a sus coordenadas en latitud y longitud, para que las leyese *Google Maps* y más tarde hiciese de *GPS*). Por último, para acabar con *Google Maps*, según el `Estado` de la Peticion y la `DireccionAHospital` de `Traslado` se selecciona a qué dirección debe ir la unidad: si al hospital o al domicilio.

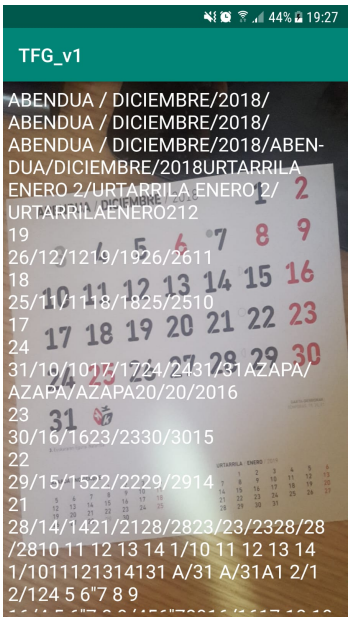


Figura 4.6: El primer OCR

A continuación, empecé a trabajar con el OCR, para leer las TS desde la cámara. En la siguiente web[5] nos detalla cómo hacer nuestro primer OCR de manera sencilla, como vemos en la figura 4.6. Para mayor facilidad, en vez de sacar una foto, tendremos un scanner constante que hace que sea más cómoda la lectura.

Utilizando la misma *actividad*, modifiqué el OCR para que procesase todas las cadenas de caracteres que le llegaban y, si alguna cadena tenía un `IndexOf(TarjetaSanitariaDelPaciente)` distinto de `-1`, la *actividad* devolvería que había encontrado el número de la TS. Si la *actividad* devuelve que ha encontrado la TS, el `Estado` de la `Peticion` se cambia.

En resumen, gracias al código obtenido de la web anterior, y la búsqueda de los caracteres procesados, obtuve un scanner que cuando encuentra el número de TS del *paciente*, finaliza la actividad e informa que se puede cambiar el `Estado` de la `Peticion` a *Completado*.

[4]Página de detalles de la migración a *AndroidX*: `https://developer.android.com/jetpack/androidx/migrate`

[5]Tutorial: como crear nuestro primer OCR: `https://www.c-sharpcorner.com/article/optical-character-recognition-by-camera-using-google-vision-api-on-android/`

Con el OCR finalizado, añadí la opción de introducir numéricamente la TS. Con esto se podían actualizar las peticiones y traslados en la BD, por ahora, local.

Finalmente, comencé a programar la *persistencia Web*. Para esta última parte, utilicé los paquetes *retrofit* y *Gson-converter*[6] de *squareup.retrofit2*. Estos dos paquetes me ayudaron a consumir Servicios API REST de manera sencilla, usando la clase `RetroFit`. Para utilizar esta clase, primero hay que definir una interfaz donde indicaremos, que parámetros de entrada necesita el servicio, donde se encuentran (header, body,etc.), cuál es su ruta y que tipo de petición HTTP es. Como, por ejemplo:

```
@GET("api/traslado/ambulancia{id}")
Call<List<Traslado>> getTraslados(@Route("id") String id);
```

En este ejemplo podemos apreciar una petición GET de HTTP a la ruta `"SERVIDOR /api/traslado/ambulancia/{id}"` donde tenemos un `String` como parámetro de entrada y una lista de `Trasldo`s como salida.

Con estas interfaces creadas podemos hacer uso de *Retrofit*, un objeto que gestionará la petición a la API REST. Para ello le pasamos como parámetro la ruta del servidor, el convertidor *Gson*, para convertir los objetos en *application/json*, y por último le añadimos un cliente para que pueda conectarse con servidores que se, consideran peligrosos o sin certificado, como el que he desarrollado. Finalmente está listo para el uso. Lo ejecutamos añadiéndole como parámetro una interfaz de la *persistencia web*, y esta clase se encargará de enviar y recibir los recursos de la API REST. En el caso en el que alguna modificación, ya sea de traslado o de petición, no pueda ser enviada, *la capa lógica* almacenará en las hojas de preferencias los traslados pendientes de envío.

Tras desarrollar la *persistencia web*, des-comenté lo que había dejado preparado en la capa *lógica* y eliminé el código que generaba entradas en la BD local. Por último creé un `ScheduledExecutorService` para que cada minuto, enviase nuestra localización al servidor que, a su vez, comprueba que no hay traslados pendientes de enviar y, en el caso de que haya alguno, lo envíe. Una vez finalizado, probé la aplicación para comprobar que todo fuese correcto.

Por último, hay que añadir que para el correcto funcionamiento de la aplicación hace falta aceptar ciertos permisos: localización, cámara, internet, y en algunos dispositivos, almacenamiento interno.

[6]El convertidor a `json` que he utilizado en *Java* se llama `Gson-converter`.

Capítulo 5

Seguimiento

En este capítulo veremos el seguimiento y el control que se ha llevado desde la semana 11 hasta la 20. La tabla que vemos a continuación es idéntica a la tabla 1.4 mostrada en el Capítulo 1. Recordemos que desde la semana 11 a la 15 corresponde a mayo, desde la 16 a la 20, a junio, y más tarde tenía dos semanas sin planificar, las cuales se utilizarían para repasar el proyecto o para aclarar cualquier imprevisto.

Tarea	**S11**	**S12**	**S13**	**S14**	**S15**	**S16**	**S17**	**S18**	**S19**	**S20**
Planificación										
Análisis										
Diseño	■	■	■	■	■					
Desarrollo					■	■	■	■	■	■
Aprendizaje y estudio	■	■	■	■	■					
Seguimiento y control		■			■					■
Redacción del documento								■	■	■

Tabla 5.1: Diagrama de Gantt semanas 11-20 de la segunda planificación

.

El seguimiento de este proyecto ha sido complejo. Cuando rescindí de mi contrato en abril estaba realizando las prácticas del máster de profesorado por las tardes, por lo que aproveché las mañanas para realizar el TFG. En dos de las semanas en las que pretendía hacer el trabajo, tuve que realizar trabajos de la asignatura de *Proyectos de Informática*, por lo que no pude avanzar mucho.

A principios de mayo, la segunda semana para ser más concretos, tuve el primer *seguimiento y control*, para la cual debía estar el análisis completo y la fase de diseño empezada. El análisis estaba finalizado, pero de la parte de diseño, solo estaba el diagrama de clases de la capa de *dominio* del servidor.

En mayo ya no tenía practicas del máster, por lo que para finales de dicho mes debía estar el diseño finalizado. A lo largo de mayo debía entregar la memoria de prácticas del máster, un documento de unas 40-50 páginas de extensión que debía contar con al menos dos unidades didácticas, información acerca del centro y obser-

vaciones de estas. Redactar este documento me llevó dos semanas. A principios de junio tenía el examen de la asignatura *Proyectos de Informática* que, por los horarios que tenía no había podido asistir a ninguna de sus clases teóricas, por lo que tenía que reservar más tiempo que el habitual para estudiarla. Para cuando llegó el control de finales de mayo, solo estaba diseñada la parte del servidor.

Al comenzar el mes de junio debía estar desarrollando el proyecto, pero me encontraba diseñando la aplicación móvil. Una vez diseñado, tuve que centrar mis esfuerzos en realizar el TFM. A este último trabajo no le pude invertir más que 18 días, por lo que no podía dedicarme a nada más.

El 25 de junio, una vez entregado el TFM, tuve que estudiar para el examen de recuperación de una asignatura del primer semestre, *Administración de Redes y Servidores*. Esta recuperación era el 29 de junio, por lo que disponía de 3 días para estudiarla.

Cuando retomé el TFG era 1 de julio. Esta semana tenía el último *seguimiento y control*, a esas alturas, según la planificación, el proyecto debía estar acabado salvo algún pequeño contratiempo. En vez de ello, el desarrollo aún no había comenzado.

El 24 de julio era la fecha límite para entregar el TFG en la convocatoria de julio/septiembre, por lo que esa era la nueva fecha límite para el desarrollo completo del proyecto. Esto hizo que para este mes tuviera una nueva planificación.

Durante esta planificación tenía nuevos hitos que ahora listaré. Hay que recordar que los días 8 y 9 de julio, no pude trabajar, ya que tenía presentación del TFM el mismo 9 de julio.

- 4 de julio: Documento redactado hasta final del diseño.
- 7 de julio: Parte servidora: *Dominio* y *persistencia*.
- 12 de julio: Parte servidora: Capa *lógica* y capa *aplicación Web*.
- 13 de julio: Parte servidora: Test unitarios.
- 15 de julio: Parte móvil: *Dominio* y *persistencia local* de la aplicación móvil.
- 17 de julio: Parte móvil: Capa *presentación*.
- 18 de julio: Parte móvil: Capa de *lógica*, pruebas de la aplicación móvil e instalación del OCR.
- 19 de julio: Redacción completa del documento.
- 21 de julio: Parte móvil: Capa *persistencia web*.
- 23 de julio: Revisión del documento final.

5.1 Horas invertidas

Como podemos ver en la tabla 5.2, las horas invertidas tienen una extensión similar a las estimadas. Quizás la más notoria es la fase de desarrollo, que me ha llevado 20 horas más de las que esperaba, pero a fin de cuentas la diferencia no es tan grande ya que he invertido un 20 % más de tiempo que el estimado.

Tarea	**Horas estimadas**	**Horas invertidas**	**Diferencia**
Planificación	25 horas	20 horas	-20 %
Análisis	25 horas	30 horas	+20 %
Diseño	90 horas	80 horas	-11 %
Desarrollo	100 horas	120 horas	+20 %
Aprendizaje y estudio	25 horas	25 horas	0 %
Seguimiento y control	10 horas	6 horas	-40 %
Redacción del documento	25 horas	25 horas	0 %
Total	**300 horas**	**306 horas**	**+2 %**

Tabla 5.2: Relación de horas estimadas e invertidas

En total he invertido aproximadamente las 300 horas, en concreto 306, con un desvío del 2 % sobre el total previsto.

Conclusiones

Durante este proyecto se han cumplido los requisitos del producto final. Me hubiese gustado disponer de más tiempo para haber desarrollado con más sosiego el proyecto, sin tanta premura, pudiendo considerar ampliaciones, como, por ejemplo, un portal web que permitiese poblar la BD, ya que, por motivos de seguridad, desde la API REST actualmente no se pueden añadir.

A lo largo de este proyecto, he podido aprender nuevas tecnologías como el desarrollo en *Android*, el uso de un OCR o el uso de *Google Maps*, y reforzar las que ya había utilizado como *Entity Framework* o la arquitectura en n-capas basada en *dominio*. Este ha sido uno de los puntos más duros y costosos del proyecto, puesto que este aprendizaje me ha llevado más tiempo del que esperaba. Considero que la fase de aprendizaje es de las más importantes, por no decir la que más, a lo largo de la vida profesional de un informático.

La comunicación con Javier ha sido excelente, desde el principio tenía claro el tipo de aplicación que deseaba y por ello pudimos hacer un alcance del proyecto y análisis de requisitos perfecto desde el inicio. No he tenido que comunicarme más que tres veces con él porque, como ya he dicho, definió perfectamente lo que quería desde el principio.

Por lo general, se podría decir que he aprendido a desarrollar un proyecto desde cero con total formalidad. Ha sido muy útil para fortalecer todo lo que he aprendido a lo largo de la carrera.